SPI master multifield HDL core

Carlos Gil Soriano
BE-CO-HT
carlos.gil.soriano@cern.ch

October 25, 2012

Abstract

A configurable SPI master multifield HDL core is depicted. Features:

- Wishbone interface.
- Support to the four operation modes.
- Three independent write fields with selectable length.
- One independent read field with selectable length.
- Protection against bad configurations.

The core is specially targeted for writing blocks of EEPROM memories which typically requiere three fields. In the case you are using a m25p32 memory, please refer to *m25p32 core*

Revision history		
HDL version	**Module**	**Date**
0.1	SPI master multifield	July 20, 2012
0.9	SPI master multifield	Sept. 20, 2012

Contents

List of Tables

List of Figures

1 Structure

The SPI module contains several blocks related the following way:

– spi_master_pkg.vhd
– spi_master_top.vhd
—– spi_master_regs.vhd
—– spi_master_slave_core.vhd
——— FIFO_dispatcher.vhd
——— gc_counter.vhd
——— gc_clk_divider.vhd

The top module combines two components: *spi_master_core* and *spi_master_regs*. The first one can be used independently from their top module, saving some interconnection lines and allowing a direct way of using the module. If access to the control registers *SPI[X]* through classic Wishbone interface is desired, then the top module must be used.

Due to the target use of this SPI core (block transfers for memory interfaces) all the three input fields are offered in both the top and the core modules.

Internally, the data in every of the three set of fields is registered by the control registers, either by directly writing into the *SPI[X]* register (in the case of *spi_master_core*) or through wishbone (*spi_master_top*).

1.1 Dependencies

Three components used in this core belong to general use in CTDAH board. Due to that they are packed inside **ctdah_lib**. The required components to be imported are:

– FIFO_dispatcher.vhd
– gc_counter.vhd
– gc_clk_divider.vhd

1.2 Operation and invalid configurations

The core analyses both SPI0 and SPI1 and if a valid operation is detected, it executes it.

Valid operations must have, at least, an instruction field with instruction length different from zero.

1.2.1 Invalid configurations

To guarantee the reliability of the core, the following invalid configurations are specified:

- A field (address write, data write or read) configured to be executed with field lenght equaling zero. That field will be skipped.
- A field with length bigger than the default length values of SPI1, will be cropped to the default length value.
- If an SPI operation is running, updates of SPI0 and SPI1 will not be attended.

2 Registers

2.1 SPI0

The SPI0 is a write-read register.

Responsible of mode of SPI operation and the length of all configurable fields (either write or read fields).

Bits	**Field**	**Meaning**	**Default**
0	**CPOL**	Clock POLarity when idle	**"00000"**
1	**CPHA**	Clock PHAse	**"00000"**
4-2	**BREAD**	Reserved	**c_READ_LENGTH**
13-5	**BDATA**	Bytes of DATA to be sent	**c_INST_LENGTH**
22-14	**BADDR**	Bytes of ADDRess to be sent	**c_ADDR_LENGTH**
31-23	**BINST**	Bytes of INSTruction to be sent	**c_DATA_LENGTH**

Table 1: SPI0 register

Values of c_READ_LENGTH, c_INST_LENGTH,_ADDR_LENGTH and c_DATA_LENGTH can be found/set in *spi_master_pkg.vhd*.

2.2 SPI1

The SPI1 is a write-read register.

Responsible of the set-up of the kind of SPI operation to be performed and the prescaling for configuring the clock of the SPI interface. Contents of internal FIFOs can be pushed to be sent, or not, to allow bigger flexibility to the user (and save power in the case in which we are sending the same contents over the SPI).

2.3 SPI2

The SPI2 register is a read-only register.

It tells when a transaction has finished and what has been sent. Due to configuration checking inside the SPI core, some fields would have been not send.

Bits	Field	Meaning	Default
0	**PUSH_DATA**	PUSH DATA bytes into internal SPI core memory	**'0'**
1	**PUSH_ADDR**	PUSH ADDRess bytes into internal SPI core memory	**'0'**
2	**PUSH_INST**	PUSH INSTruction bytes into internal SPI core memory	**'0'**
5-3	x	Reserved	**"000"**
6	**READ_MISO**	READ bytes from MISO line	**'0'**
7	**SEND_DATA**	DATA bytes will be sent in a write operation	**'0'**
8	**SEND_ADDR**	ADDR bytes will be sent in a write operation	**'0'**
9	**SEND_INST**	INST bytes will be sent in a write operation	**'0'**
10	**SEND_OP**	perform a SEND OPeration	**'0'**
11	y	Reserved	**"00"**
15-12	**CLK_DIV**	CLocK DIVider	**X"0"**
31-16	z	Reserved	**X"0000"**

Table 2: SPI1 register

Bits	Field	Meaning	Default
0	**MISO_DUP**	MOSI Data UPdate	**'0'**
1	**READ_DONE**	READ process DONE	**'0'**
2	**SENT_DATA**	DATA was SENT	**'0'**
3	**SENT_ADDR**	ADDRess was SENT	**'0'**
4	**SENT_INST**	INSTruction was SENT	**'0'**
5	**SENT_OP**	OPeration was SENT	**'0'**
11-6	x	Reserved	**X"00"**
15-12	**CLK_DIV**	CLocK DIVision	**X"0"**

Table 3: SPI2 register

2.4 SPI3

The SPI3 register is a read-only register.

It holds the data received by the MISO pin. Valid data can be read as soon as *rd_SPI3_o* output in *spi_master_core.vhd* goes high.

Bits	**Field**	**Meaning**	**Default**
31-0	**MOSI_DATA**	Latest 32 bits received	**X"00"**

Table 4: SPI3 register

3 Internal memory mapping

The internal registers map over the wishbone interface is as follows:

Address	**Register**	**Access**
0x0	*SPI0*	Write-read
0x1	*SPI1*	Write-read
0x2	*SPI2*	Read-only
0x3	*SPI3*	Read-only

Table 5: Memory mapping

4 How to use it

4.1 Perform an operation over SPI

It consists on writing the SPI0 register first, and then the SPI1 register. **Order must be preserved**.

Status of the operation can be followed via SPI2 register. SPI3 register offers the data read from the MISO line.

4.1.1 SPI0 register

It should be specified:

- Mode of operation (via *CPOL* and *CPHA*).
- Number of bytes to write (instruction *BINST*, address *BADDR* and data *BDATA*) into MOSI line.
- Number of bytes to read (*BREAD*) from the MISO line.

Those values must be written into the SPI0 register and, then, the same for SPI1 register.

4.1.2 SPI1 register

The following fields must be set up:

- Which contents of the internal FIFOs (instruction, address and data) should be loaded into the SPI core to be sent over MOSI line: *PUSH_INST*, *PUSH_ADDR* and *PUSH_DATA*. If not pushed, last pushed value will be written.
- Clock divider, *CLK_DIV*, to be generated the SPI clock frequency from the system clock frequency.
- Which fields should be written: *SEND_INST*, *SEND_ADDR SEND_DATA*.
- Select if a read operation will be carried out: *READ_MOSI*.
- Determine that an operation should be consider by the core: *SEND_OP*

Note that invalid operations will be detected by the core (either they will be rejected or modified to comply with the HW constraints).

4.2 SPI2 register

When a SPI operation has finished, a one clock signal (SENT_OP bit) is flagged for one system clock. During this one clock signal, all the fields that have run in the operation can be checked in *SPI2*.

For optimum perfomance, polling of SENT_OP should be carried out.

4.3 SPI3 register

SPI3 has the contents of the MISO read values.

It can be read in any moment but operation-consistent information can be obtained when the SPI operation has finished.

A IP core insights

A.1 Design structure

The core is governt by a simple finite state machine, *clk_fsm*, and a expansion of one of it states, *spi_clk_fsm*.

A.1.1 clk_fsm

The main functionality of clk_fsm is:

- To place all the signals of the core to a known default state by means of a reset state (*R0_RESET*, in red).
- To provide the VHDL predifine set-up and hold times of the chip select line. This functionality is specially important in the case of the hold time to let some memories to be programmed correctly when continuos writes are performed into them (*S1_SETUP* and *S3_HOLD*, both in grey).
- To access to SPI slaves (*S2_SPI_ACTIVITY*, in golded-yellow).

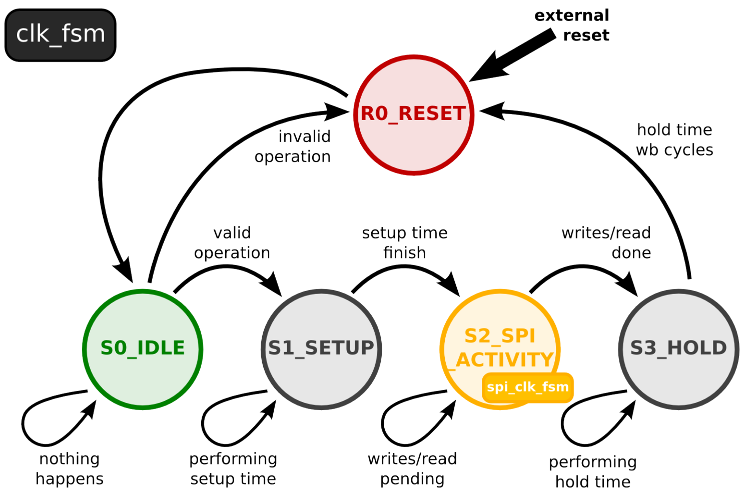

Figure 1: clk_fsm

A.1.2 spi_clk_fsm

Namely the expansion of clk_fsm's state *S2_SPI_ACTIVITY*, it is responsible of controlling the writes (including memory prefetching) and reads of the MOSI and MISO lines, respectively.

When clk_fsm goes into *R0_RESET* state, spi_clk_fsm resets to *S0_IDLE* and it is ready to act over the SPI interface. When a valid instruction is present and clk_fsm's *S1_SETUP* has change to *S2_SPI_ACTIVITY*, the refresh of internal FIFO memories is carried out (according to valid fields to be sent and PUSH_[x] values in SPI1 register. Then, in the case of write fields the byte to be sent is pulled from the FIFO (in blue) and subsequently sent. In the case of reads, SPI data is read and a one-clock byte reception is issued (in blue). After that two states for clearing up all the values are run.

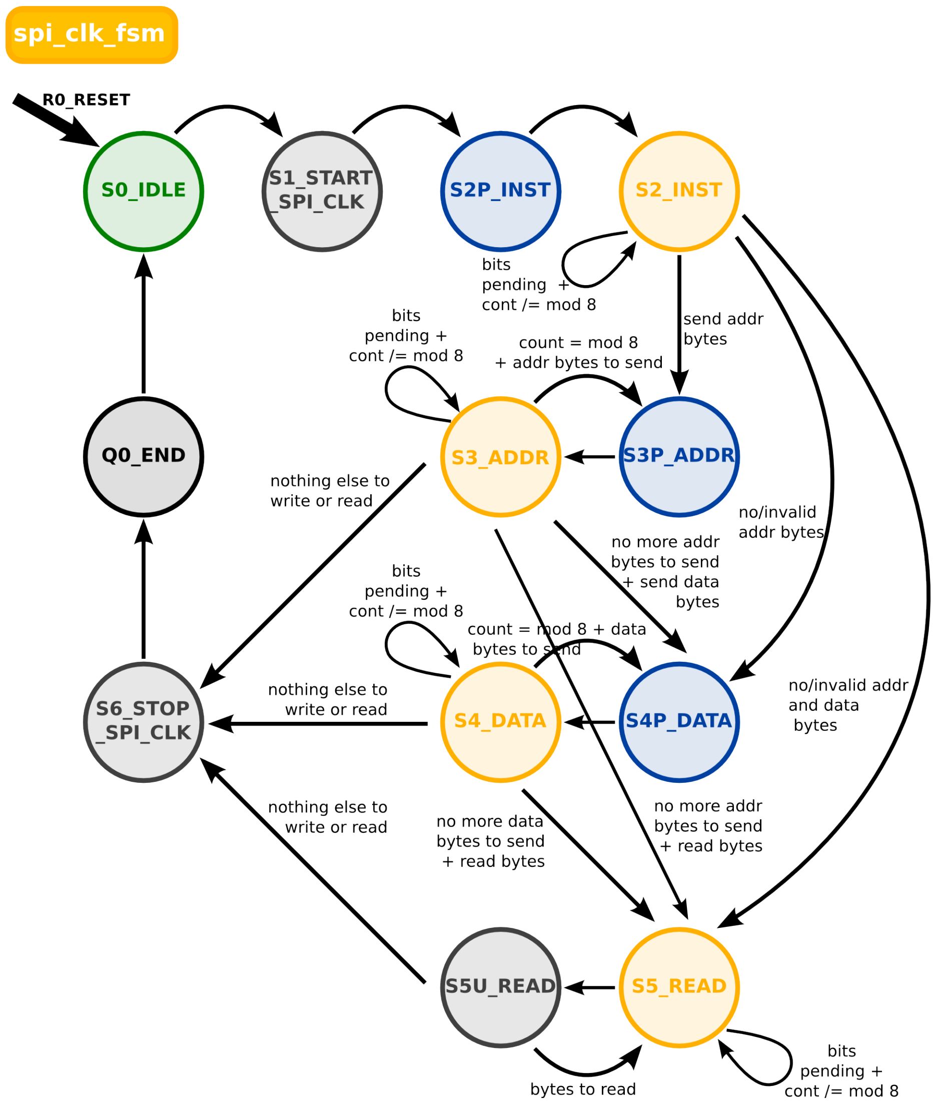
spi_clk_fsm
R0_RESET
S0_IDLE
S1_START _SPI_CLK
S2P_INST
S2_INST
bits pending + cont /= mod 8
send addr bytes
count = mod 8 + addr bytes to send
S3_ADDR
S3P_ADDR
Q0_END
nothing else to write or read
no/invalid addr bytes
no more addr bytes to send + send data bytes
count = mod 8 + data bytes to send
S6_STOP _SPI_CLK
S4_DATA
S4P_DATA
no/invalid addr and data bytes
no more data bytes to send + read bytes
no more addr bytes to send + read bytes
S5U_READ
S5_READ
bytes to read

Figure 2: spi_clk_fsm